Android Root and its Providers: A Double-Edged Sword

Hang Zhang, Dongdong She, Zhiyun Qian
University of California, Riverside
hzhan033@ucr.edu, sdongdong@engr.ucr.edu, zhiyunq@cs.ucr.edu

ABSTRACT

Android root is the voluntary and legitimate process of gaining the highest privilege and full control over a user's Android device. To facilitate the popular demand, a unique Android root ecosystem has formed where a variety of root providers begin to offer root as a service. Even though legitimate, many convenient one-click root methods operate by exploiting vulnerabilities in the Android system. If not carefully controlled, such exploits can be abused by malware author to gain unauthorized root privilege.

To understand such risks, we undertake a study on a number of popular yet mysterious Android root providers focusing on 1) if their exploits are adequately protected. 2) the relationship between their proprietary exploits and publicly available ones. We find that even though protections are usually employed, the effort is substantially undermined by a few systematic and sometimes obvious weaknesses we discover. From one large provider, we are able to extract more than 160 exploit binaries that are well-engineered and up-to-date, corresponding to more than 50 families, exceeding the number of exploits we can find publicly. We are able to identify at least 10 device driver exploits that are never reported in the public. Besides, for a popular kernel vulnerability (futex bug), the provider has engineered 89 variants to cover devices with different Android versions and configurations. Even worse, we find few of the exploit binaries can be detected by mobile antivirus software.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security, Measurement

Keywords

Android root exploit, root provider

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2810103.2813714.

1. INTRODUCTION

We are in an age when customers are not given full control over the purchased personal mobile devices such as smartphones and tablets. Due to the popular demand by users, a unique ecosystem of offering smartphone root or jailbreak has formed. Root and jailbreak are the process of obtaining full privilege on Android and iOS devices respectively. They allow users to bypass restrictions set by carriers, operating systems, and hardware manufactures. With full control over the device, a user can uninstall bloatware, enjoy the additional functionalities by specialized apps that require root privileges, or run paid apps for free.

Classified by whether a device is flashed, there are two types of root methods: 1) soft root. 2) hard root. The former refers to the case where root is obtained directly by running a piece of software (*i.e.,* root exploits). The latter refers to the case where *su* binary is flashed externally via an update package or ROM. Depending on the device model and OS version, different root methods may be applicable. For instance, due to locked bootloaders, some devices cannot use hard root. Similarly, if a particular device has no software or hardware vulnerabilities whatsoever, soft root would not be possible. In practice, like any other systems, Android devices do have a variety of vulnerabilities in various components: kernel, driver, and application as summarized in §2.

In this paper, we focus on the soft root as the same exploits can be potentially abused by malware authors and therefore much more dangerous than hard root. In Android, such root service is provided by a number of parties. Individual developers or hackers often identify vulnerabilities, develop, and publish exploit tools to gain fame and possibly fortunate. However, due to the diversity of Android devices in terms of hardware, fragmented OS versions, and vendor customization [34], it is simply not scalable for individuals to engineer a large number of exploits to cover a wide range of the devices. Therefore, the business of offering root as a service has emerged [10, 8, 13].

Interestingly, most commercial root providers are free to use. They operate by requiring the exploit to run on an Android device by a user voluntarily, *e.g.,* through an one-click root app [10, 8, 13]. Unfortunately, attackers can also acquire such exploits easily by impersonating a regular user. To make the problem worse, some of the large root providers have a large repository of root exploits or even invent new ones so they stay ahead of their competitors. This may give attackers a strong incentive to target such providers.

In this paper, we examine the root ecosystem closely to understand the following high-level questions: 1) How many

types and variations of Android root exploits exist publicly and how they differ from the ones in commercial root providers. 2) How difficult is it to abuse the exploits offered by the root providers. We answer the above questions by undertaking a series of measurement and characterization of root exploits as well as the providers that offer them.

The contributions of the paper are the following:

- We conduct a comprehensive measurement study on Android soft root methods to understand their origin and overall trend. We find that 1) most public Android root exploits target the application-layer vulnerabilities that affect only specific types of devices. 2) Although kernel vulnerabilities are considered the most dangerous, an exploit developed on one device may need to be adapted to work on another. 3) As kernel vulnerabilities become rare, device drivers become the dominating target to find root exploits.
- We analyze the security protections employed by a number of root providers on their exploits. While larger root providers often employ more protections, we identify systematic weaknesses and flaws which substantially undermine their effort. The result calls for better security practices on protecting such dangerous exploits.
- We survey the availability and variety of the exploits online versus the ones extracted from root providers which range from large security companies to individual developers. We report that a large root provider not only keeps "secret" exploits, but also spent significant engineering effort to polish and adapt existing exploits.

2. PUBLICLY AVAILABLE ANDROID ROOT EXPLOITS

In this section, we attempt to exhaustively collect all publicly known Android root exploits or vulnerabilities and understand their characteristics. Even though root exploits are reported to have been used by malware in the wild already [43, 28, 19], we still lack a complete and up-to-date picture. We are not aware of any systematic research studies on root exploits used by malware in the recent two years. It is unclear what exploits may be currently used and which ones are easily usable by malware, thus likely to appear in the future. We aim to understand the question by analyzing the current and publicly available resources to malware authors.

Data sources and collection methodology. We survey a large number of public sources including academic papers [30], research projects [1], published books [26], as well as online knowledge base(*e.g.,* CVE database or forum post such as XDA forum) [9, 6, 2, 5]. Search terms including "Android root", "root exploit", and "privilege escalation" are used to locate the relevant information. Note that even though we attempt to collect an exhaustive list based on our expertise, it is inherently a best effort. The list eventually leads to a dataset of 73 exploits or vulnerabilities.

In most cases, a vulnerability (with a CVE number) maps to a corresponding exploit. However, as will be described in §2.2, we are unable to locate publicly available exploits for some small subset of CVEs. In many other cases, the opposite is also possible — no CVE number is assigned but the exploit is readily available, likely because of its limited impact on very few device types.

We also observe that some exploits require multiple vulnerabilities to gain root. For instance, master key vulnerability (*e.g.,* ANDROID-8219321) only leads to system user privilege. Additional vulnerabilities are necessary to complete a root exploit [16]. In such few cases, we consider them two separate ones. In the survey, we found 5 vulnerabilities which can gain system privilege only and 3 vulnerabilities which can gain root permission from system privilege (we count them as 8 still). In addition, some exploits are related to each other and can be considered variations of one another. When it is possible to locate different CVE numbers or vulnerabilities through the technical details, we also consider them different exploits. This inclusive strategy is more likely to lead to a more complete discovery of exploits and vulnerabilities.

At a high level, we are able to locate enough technical descriptions of the vulnerabilities or exploits, although they vary significantly in detail, clarity, and availability of source code or binaries. Perhaps expectedly, we find that the majority of the exploits are not produced by academic research.

Questions to answer. We aim to answer the following questions from the analysis:

1. How many general vs. specific exploits exist? Intuitively, some exploits are more general than others, especially those exploiting kernel vulnerabilities. Some exploits may be applicable to certain vendors only.
2. Whether the exploit source code or binaries are publicly available? What's the requirement to run the exploit? Can the exploit work via a standalone app, *e.g.,* without user intervention or booting into recovery mode?
3. Whether antivirus can recognize the exploits?

2.1 Root Exploit Impact and Coverage

To understand the impact and coverage of an exploit, we first try to identify the layer that is targeted. This is because the impact could be different depending on the layer. For instance, if it is a vulnerable setuid program (in the application layer) installed only a certain models of a vendor, then its impact will be limited. We divide the layers into four categories based on the Android Architecture:

Linux Kernel. Due to its privileged position, targeting Linux Kernel is natural to achieve full control over an Android device. In particular, a vulnerability in the kernel has a large impact as all devices that run the vulnerable kernel can potentially be affected. For instance, TowelRoot (CVE-2014-3153) exploits the *futex* syscall bugs to gain root access and it is considered to affect all kernel versions before 3.14.5. In this category, we include everything running inside kernel except the cases described below.

Vendor-Specific Kernel or Drivers. Different from the main kernel code that runs on almost every device, vendors either customize the kernel (*e.g.,* Qualcomm's custom Linux kernel branch) or provide vendor-specific device drivers for various peripherals (*e.g.,* camera, sound) [42], Such code runs inside the kernel space and the compromise of which can also lead to full control over the device. Given that they are produced by a single party without much open auditing, and sometimes closed source (*e.g.,* especially the device drivers), the chance of them having security vulnerabilities can be high, as confirmed in our measurement results. However, since not all Android devices run the same set of customized kernel or device drivers, an exploit on a specific customized device can only impact a subset of Android devices (*e.g.,* certain Samsung devices).

Libraries Layer. Exploits at the libraries layer target the Android libraries or external libraries used for support-

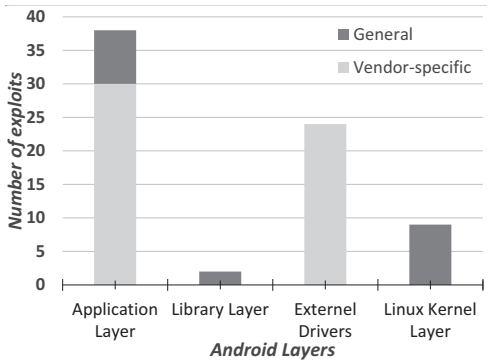

Figure 1: Number of exploits by layer

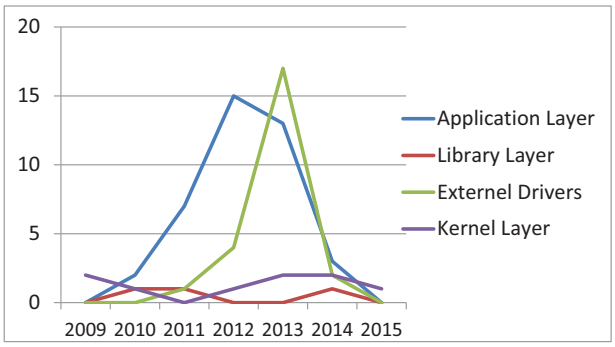

Figure 2: Number of exploits by year

ing different applications. For instance, in ZergRush exploit (CVE-2011-3874), libsysutils used by Volume Manager daemon (running as root) in Android is shown to have a stack overflow vulnerability that leads to root privilege escalation [26]. The vulnerability in such libraries can have a large impact because they may be embedded by multiple programs, as long as one such program runs with root privilege and exercise the vulnerable code, a root exploit can be successfully constructed. The ObjectInputStream vulnerability (CVE-2014-7911) is another example.

Application and Application Framework Application layer root exploits mostly include vulnerable logics introduced by setuid utilities, system applications, or services. The impact of such exploits depends on whether it is a third-party one or not. So far, most cases are from third-parties which indicate a limited impact. One example is a vulnerable setuid utility that is only present on XoomFE devices that has a command injection vulnerability [21]. Another instance is a backdoor-like setuid binary shipped with certain ZTE Android devices (CVE-2012-2949).

In general, from the highest to lowest, the order of impact and generality of exploits would be 1) the kernel exploits, 2) the exploits targeting libraries that are used by Android system processes, 3) exploits targeting system applications or services, and 4) exploits against vendor-specific device drivers, applications, and programs.

Even though we cannot accurately predict the number of devices impacted by each exploit, the reasoning is that kernel and Android system code is much more widely used than the vendor-specific code. In addition, patches of kernel vulnerabilities are much harder to reach the end-user whereas application updates can be quickly pushed out.

Breakdown by layer. As shown in Figure 1, out of 73 exploits, there are 54 exploits that are vendor-specific (in lighter gray) and 19 general exploits (in darker gray). The vendor-specific ones include all device driver exploits, and vendor-specific applications or programs. The *application layer* is found to have the largest number of exploits, although most of them are vendor-specific. The *external drivers layer* has the second largest number of exploits but all of them are vendor-specific by definition. It is expected that the *kernel layer* and *library layer* have the smaller number of exploits which are very general and extremely dangerous. The number of new *kernel layer* exploits occurred each year is also relatively stable in our survey – only one or two per year on average.

Time dimension. Another important dimension is time. Specifically, the lifetime of vulnerability is determined by the patch version and date. The later the discovery, the longer the vulnerability lives and therefore a higher impact. On the other hand, the sooner the discovery, the more quickly the root exploits can be developed. Figure 2 shows the number of exploits discovered in each year. As we can see, vulnerabilities explode around year 2013 due to a large number of vulnerabilities introduced by vendor customization at the *external drivers layer* and *application layer*. One of the key problems is that device files on many vendor-customized Android systems have weak permissions [42], without which an app process cannot even open these device files and launch exploits. Such period also coincides with the increased market share of Android and participating vendors. On the other hand, the kernel and library layers have a relatively stable pattern.

Obviously, with common mistakes corrected in the vendor customization process, the number of vendor-specific vulnerabilities will drop. However, it is always hard to predict the new trend or classes of exploits that may surface. As long as there is strong need from users, we believe root exploits will still continue to exist in the foreseeable future. For instance, at the time of writing, a new kernel-level root exploit named PingPong root [40] is announced.

Coverage. Theoretically, a kernel vulnerability affects all kernel versions between when the vulnerability is introduced and when it is fixed. Therefore, a recently discovered kernel vulnerability such as TowelRoot (CVE-2014-3153) should have a significant coverage. However, as will be discussed in §3, it is most often not the case. In practice, a kernel exploit may depend on system configurations, address space layout, compiler options, *etc.*. Therefore, to successfully root a device, multiple exploits are usually attempted in both the malware [43] and the root providers.

2.2 Exploit (Source or Binary) Availability

In this section, we aim to understand how readily available the exploit source code or binaries are on the Internet for public use. In particular, the availability is a direct indication on whether malware authors can find and leverage such exploits. Even though it is well known that malware already start to embed root exploits that are often copied from the public sources [43], it is unclear how many such exploits can be located and abused.

To locate exploit source code or binaries, the methodology is to simply use the relevant keywords (when applicable) of an exploit that typically include the CVE number (*e.g.,* CVE-2014-3153), the Google Bug ID (*e.g.,* ANDROID 3176774), impacted device model, and the exploit nickname

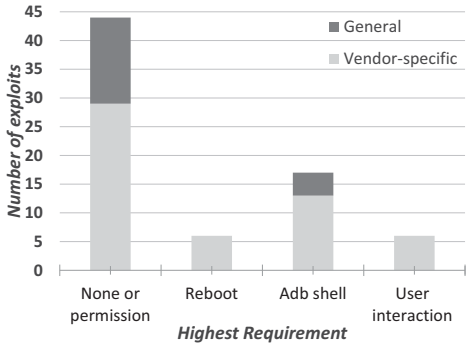

Figure 3: Exploit requirement breakdown

Root exploit	AVG	Lookout	Norton	Trend Micro
exploid(2010)				
Zimperlich(2010)	X	X		
Gingerbreak(2011)	X	X	X	X
BurritoRoot(2012)	X	X		X
Poot(2013)				X
LGPwn(2013)			X	X
WeakSauce(2014)	X	X		
Framaroot(2014)	X			X
Towelroot(2014)	X	X	X	X
PingPong root(2015)				X

Figure 4: Detection results of mobile antivirus

(*e.g.,* TowelRoot). To ensure adequate coverage, we undergo two rounds of independent web searches.

Out of the 73 cases, we are able to locate either the source code or binary of 68 exploits. Only 5 of them have neither found. One of them is not available because it is only described in a research abstract. Others are not available even though the corresponding CVEs clearly indicate they allow arbitrary code execution with elevated privileges. We are not certain about the root cause but one plausible explanation is that the person who discovered the vulnerability did not release the technical details or any proof-of-concept exploits. It is also possible that the vulnerabilities are not generic enough to attract individual hackers to build an exploit.

Theoretically, both the binary and the source code are valuable to malware authors. A malware can embed the binary directly so long as it is an independent piece (*e.g.,* executable or libraries) and has an easy-to-identify interface. Of course, source code has many advantages since it can be freely customized and improved.

Overall, there are 46 exploits with source code available, 18 of them are simple exploits that leverage weak file permissions and symbolic link attacks [14] which are typically introduced by vendor customization. Such exploits can be mostly implemented in shell scripts. The rest are written in C (and one in Java against CVE-2014-7911). On the other hand, there are 22 exploits with binaries available only, which are in the following two forms: 1) PC-side scripts that may push additional binaries onto the device and 2) apk files that run on the device directly. There are 10 and 12 of them respectively.

We observe that even though source code is generally more valuable, it may not be as robust as the binaries, especially when the source code is offered as "third-party" proof-of-concept. Particularly, in order to accommodate different devices and models, considerable iterations and engineering efforts are required. For instance, TowelRoot is binary only and it has evolved over three major revisions supporting different devices. The available source code, however, is just proof-of-concept and is written by other developers [4].

To summarize, malware will likely be able to integrate most of the exploits, even if some may have limited coverage.

2.3 Exploit Requirements

Even if an exploit source or binary can be found, one still needs to understand the requirement to run them. For instance, an exploit may require an adb shell setup through a PC connection — since only processes running as *shell* user can perform the exploit. In other cases, an exploit may require user interactions (*e.g.,* booting into recovery mode at least once to trigger the vulnerable code).

To understand the exploit requirements, we perform two steps: 1) locate technical reports or tutorials published either by the exploit authors or other interested parties [21]. 2) if 1) is not available, we attempt to read the exploit source code or script, which will typically contain such information. It turns out the two steps can cover most exploits.

From the technical details of the exploits and source code, we are able to identify the following major requirements (from the most rigid to the least):

- **Requiring user interactions.** This category have few cases. One case is asking the user to download an app and manually interrupt the installation [7]. One is asking the user to boot into recovery at least once [18]. Another is asking the user to manually put the device into "battery saving" mode [12]. The last asks the user to open a vendor-specific app and hit a button [15]. Intuitively, exploits in this category are difficult to be used by malware authors to fully automate the exploit.
- **Requiring adb shell through a PC connection.** For some exploits, adb shell connection is required because of the following most common reasons: 1) The exploit can successfully modify a setting in local.prop which enables root for adb shell only. 2) The exploit needs to write to a file owned by group *shell* and group-writable (not world-writable) [14]. 3) The exploit targets the adb daemon process that requires the attack process to run with *shell* user. For instance, the Rage Against the Cage exploit [11] targets the vulnerability of adb daemon's missing check on return value of setuid().
- **Reboot.** Generally, many root exploits require at least one reboot. For instance, a symbolic link attack would allow an attacker to delete a file owned by *system* with weak permission, *e.g., /data/sensors/AMI*304*_Config.ini*, to setup a link at the same location to a protected file, *e.g., /data/local.prop*. After a reboot, the corresponding init scripts would attempt to change the permission of the original file (*i.e., /data/sensors/AMI*304*_Config.ini*) to world-writable, which in reality changes the permission of the linked file (*i.e., /data/local.prop*).
- **None or permission.** The exploits in this category have no hard requirements, however, some of them may require certain Android permissions like *READ_LOGS* in order for the process owner to be placed in certain user group.

If an exploit has multiple requirements, we will only count the most rigid one (*e.g.,* one exploit needs both adb shell and reboot will fall into "adb shell" category). The results are

summarized in Figure 3. 6 of them require user interactions. 17 of them require adb shell through a PC connection. 6 of them require rebooting the device. The rest 44 do not have any hard requirements, in which 5 do require certain Android permissions. Most exploits are vendor-specific, however, there are 4 and 15 general exploits in "Adb shell" and "None or permission" categories, respectively.

Correlating with the 68 exploits that have source code or binaries, there are 39 available exploits that need only reboot, permission, or none and can potentially be abused in a malware silently gaining root access.

2.4 Root Exploits Detection by Anti-Virus

Since root exploits can be potentially abused by malware to gain the highest privilege, we expect they are of high priority to antivirus software. To verify the hypothesis, we download 21 root exploits in the form of 10 apk files or ARM ELF executables. In order for the antivirus to recognize a malicious app, when there is a lack of apk file, we package the ARM ELF executables into a simple Android app (stored in the libs folder). Since some source code is only proof-of-concept, we decide to use the binary only for experiment.

We downloaded 4 antivirus software from the Google Play: AVG AntiVirus Free 4.3.1.1.213361, Lookout Security and Antivirus (lookout) 9.18.1, Norton Mobile Security V3.10.0.2361, Trend Micro Mobile Security V6.0.1.2050.

We use a Galaxy S3 phone to carry out the experiment in the early May of 2015. We install the antivirus software one at a time and never keep two or more running simultaneously to prevent potential conflicts. The antivirus software all have real-time protection enabled and will pop up a window when they believe a malware or suspicious app is detected.

Table 4 shows the results with exploits ordered by year. Note that in the case of Framaroot, 12 exploits are packed in the same apk, and only one row is shown to represent them. As we can see, most exploit binaries are flagged by more than one antivirus software, which is expected as most of them are well-known root exploits.

There is one exploit, exploid that cannot be recognized. Poot and PingPong root are detected by one antivirus only, indicating that some exploits can still fly under the radar(The reason may be that our samples are different from those used by malware). Interestingly, the very recent PingPong root exploit, published a few days ago at the time of writing (early May 2015), is already detected by Trend Micro, indicating that they are specifically sensitive about the publicly available exploits.

In contrast, as will be shown in §6.3, the exploit binaries engineered by large root providers are surprisingly "clean" as all major antivirus software have difficulty detecting them.

3. ADAPTATION OF ROOT EXPLOITS

vspace0.03in Android is well known for its fragmentation due to carrier and vendor customization [34]. On one hand, the availability of a large number of customized Android devices allow greater market penetration. On the other hand, the diversity of Android devices makes it extremely difficult to write robust root exploits that work on devices with varying application/kernel configurations and settings.

It is known that many exploits require adaptations to work across devices. In fact, it is believed that adaptations directly discourage the malware authors to use certain exploits (*e.g.,* zergrush and mempodroid) in the wild [28].

Typically, an exploit needs specific environment to work. Difference in CPU, kernel version, OS version may cause a failure. In memory corruption based exploits, requiring the knowledge of absolute or relative memory addresses of certain key data structures are common reasons why adaptation may be necessary. For instance, in the exploits against CVE-2014-4322, the address of a kernel symbol needs to be determined which differs across devices. Note that a bruteforce approach will not work as it may crash the system jumping to an undesirable address. In rare cases such as Exynos-abuse, adaptations may not be necessary. This exploit can access arbitrary physical memory through a vulnerable device driver and overwrite kernel data to gain root privilege. Instead of using a hard-coded static address, the exploit can search from the beginning of kernel space to locate the target kernel symbol. Of course, such a special vulnerability allowing the whole kernel space access is unlikely to occur and is limited to certain device type only. In general, kernel exploits require substantial adaptation, as is demonstrated through the case study below.

3.1 CVE-2014-3153 (Futex bug)

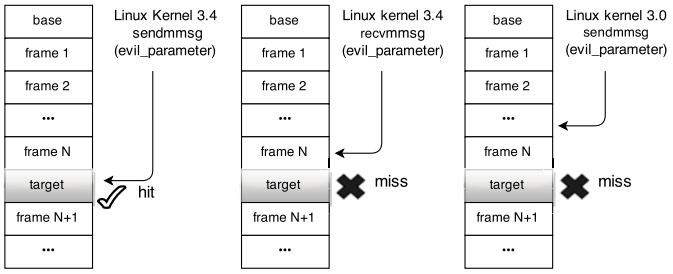

Figure 5: CVE-2014-3153: kernel stack overwrite by invoking system calls

The futex kernel vulnerability is reported to affect all kernel versions prior to Jun 3rd 2014 and its first exploited by TowelRoot. It was originally designed to root Verizon Galaxy S5, then modified to be compatible with more devices, including ATT Galaxy S5,Galaxy S4 Active, Nexus 5. Although it claims to possibly work on every android device with a vulnerable kernel, a slight variation of hardware platform or kernel versions may cause this exploit with high precision requirement to fail. To cover more devices, it adds a feature named mod strings for users to modify 5 different exploit variables. We explain one of the key variables, system call, in detail below.

The system call variable specifies one of the four possible system calls utilizable to carry out part of the exploit. The context is that the attack sets up a pointer in kernel heap to point to a kernel stack address that is subject to overwrite by a system call. As shown in Figure 5, an attacker needs to pass a malicious parameter to a system call, which will be copied into the kernel stack, and hope that it will eventually land on the target address in kernel stack. Depending on the exact kernel version and configuration, there are two obstacles: 1) the target stack address may be different relative to the base stack address. 2) the depth that the malicious system call parameter reaches can also be different. We illustrate such obstacles in the figure. In the first case, the parameter of syscall `sendmmsg()` can be successfully placed to overwrite the target stack address. In the second case, however, due to the wrong syscall chosen, the

Name	Components	Devices supported (claimed)
Root Genius	PC/MOBI	20,000+
360 Root	PC/MOBI	20,000+
IRoot	PC/MOBI	10,000+
King Root	PC/MOBI	10,000+
SRSRoot	PC	7,000+
Baidu Root	PC/MOBI	6,000+
Root Master	PC/MOBI	5,000+
Towelroot	MOBI	N/A
Framaroot	MOBI	N/A

[1] **PC**:PC-side software **MOBI**:Android app

Table 1: Root Providers List

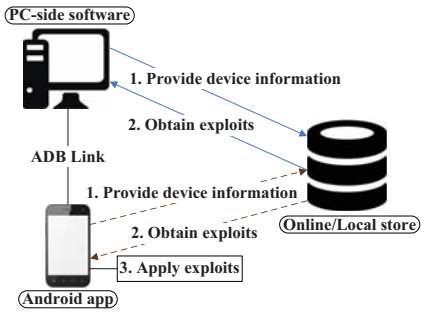

Figure 6: General Root Architecture

parameter may fail to hit the target address. In the third case, the same syscall `sendmmsg()` is chosen, but due to kernel version difference, the reached depth is different, missing the target. Due to the above difficulties, Towelroot suggests users try different syscalls in combination with other variables to hope that the target address will be hit.

It practice, we have tried the Towelroot on 3 devices that we can get our hands on, all with a vulnerable kernel built prior to Jun 3rd, 2014, yet Towelroot fails to root 2 of them.

4. ROOT PROVIDERS OVERVIEW

As alluded before, there exist a large number of root providers, ranging from ones developed by individuals to large companies. In this section, we aim to study them in the following aspects: 1) survey different types of root providers and understand how they operate. 2) characterize the protection strength on the carried dangerous root exploits. 3) measure the extracted providers' exploits and understand their relationship with the publicly available ones.

In general, the discrepancy of the available resources between small and large providers is likely a key factor in deciding the above aspects. There are a number of popular root providers which contain both the largest and newest ones as listed in Table 1. We confirm that the larger providers do offer a much more comprehensive set of exploits, however, even though present, the corresponding protections are substantially far from being adequate. In fact, we find serious weaknesses that allow us to extract and study a large portion of the exploit binaries from one large provider.

We study 7 out of the 9 providers in depth and anonymize their names for security concerns, as shown in Table 2.

Methodology and collected results. We collect information of three main categories: 1) Public information about each provider, *e.g.,* number of devices that they claim to support, whether it has a PC-side program and/or an independent Android app, as shown in Table 1. 2) Exploit information including the location of the binaries (*e.g.,* on a remote backend or local), and the quantity of them. 3) Protection employed by root providers to prevent the exploit binaries from being reverse engineered and abused by others. This gives a rough estimate on the level of difficulty to extract the valuable exploits for malicious purposes. The information collected in 2) and 3) requires understanding of the inner workings of providers via reverse engineering.

Root provider architecture. From all the providers we studied, a common architecture is depicted in Figure 6. The service is typically through either a PC-side program and/or an independent Android one-click root app. The former can control a device via the ADB interface and thus utilize both ADB-related exploits like "rage against the cage" and others launched directly on the Android device. The latter Android app can operate independently to execute root exploits.

The main program logic involves three key steps:

STEP 1: Collect device information such as model name, kernel and Android version, hardware platform and so on.

STEP 2: Based on the information, obtain proper exploits from either remote servers or local store.

STEP 3: Execute the chosen exploits on the device to gain root permission.

As shown in Table 1, the providers are sorted by their coverage. It is obvious that all larger root providers are comprehensive in offering both the PC-side program and the independent Android app, whereas smaller providers typically offer exploits in one way or the other.

Number of exploits. Here we focus on the exploits that can be launched directly through the Android app, since they are much more likely to be abused once stolen by attackers. In Table 2, we list the number of exploit binaries we are able to locate for each provider. It is surprising to see that the number goes over one hundred for the largest one. In addition, the number is only a lower bound as there can be others that we are not able to find (See §5.1 for details on how to locate the exploits). Such number is significantly higher than what we can find from public sources, highlighting the potential risk of being targeted by attackers. Note that we sort the table by the number of exploits we can find, yet it does not correspond to the same order presented in Table 1, therefore, not revealing the provider names.

We do realize that different providers may organize their exploits differently into binaries. One binary could correspond to a single exploit with or without its variants. Therefore, simply counting the number of exploit binaries can be biased. In §6, we offer a more comprehensive analysis on the exploits and compare them with the publicly available ones.

Protection Strength. Perhaps expectedly, we observe that larger root providers with more exploits tend to employ stronger protection of their products, and smaller providers usually employ little to no protection. For instance, as shown in Table 2, provider 1 and 2 not only protect the Android one-click root apps, but also introduce tamper-detection and encryption in their exploit binaries (typically in native code) to prevent them from being stolen and abused. In addition, the network communication to retrieve exploits from its remote store is also encrypted. In contrast, provider 6 and 7 only equip some basic protection in their exploits, which is easy to bypass. In the study we also find that some larger providers will integrate small providers' apps or exploit binaries directly, this observation again reflects the

NO.	Exploits		Protection			
	Store	Amount	Store	PC-side	Device-side	Exploits
1	LOC/OL	160+	OL	A	ANP*	COPS
2	LOC/OL	60+	OL/LOC	N/A	NO	CS
3	LOC	40+	LOC	None	ANP	S
4	LOC	20+	LOC	N/A	O	CS
5	LOC	20+	None	P	N/A	None
6	LOC	10+	None	N/A	None	C
7	LOC	<10	None	N/A	None	O

[1] *: Not true for its app from a special channel

Table 2: Root Providers Measurement Result. **OL**:Online **LOC**:Local **N/A**:Not applicable or studied. **A**:Anti-debug **C**:Tamper-detection **N**:Code Protection with JNI. **O**:Obfuscation **P**:Packing **S**:String Encryption.

lack of protection for small root providers which are usually individual hobbyists. Unfortunately, as we will show in §5.1, the seemingly strong protections in large providers can in fact be broken down fairly quickly due to several severe weaknesses we identify.

PC-side vs Device-side Protection. It is important to realize the PC-side program and independent Android app contain duplicate functionality of reading device information and retrieving exploits from local or remote store. Therefore, as long as either one has a weak protection, the procedure can be revealed and exploits maliciously retrieved. Indeed, security is only as strong as its weakest link.

Interestingly, we observe the protection strength is indeed typically inconsistent. Compared to PC, Android has a much shorter history which results in fewer available commercial-grade protection methods, *e.g.,* VM-based protection [23]. This is supported by our finding that most providers have weaker protections on its Android app compared to its PC counterparts, even though they usually throw in a number of protections, hoping that they are strong enough (*e.g.,* ProGuard). On the other hand, provider 3 does employ a stronger commercial protection solution called Bangcle on its Android app, yet it has no protection whatsoever on its PC program. The result is summarized in Table 2. In the cases of a "N/A", it indicates that we did not study it since the other side can already be successfully reverse engineered.

Besides above observation, extra opportunities exist where inconsistently protected software may be distributed. First, older versions and newer versions of the same software may implement the same core functionality, but stronger protections are added only to the newer versions. For instance, we observe an old version of an Android one-click root app from a provider has significantly weaker protection than the new ones. Second, in rare cases, some root providers may share code with each other, yet one version may have much weaker protection than others. We observe one such case among the providers - the involved two providers have a cooperating relationship.

5. CASE STUDIES OF PROTECTION MECHANISMS

Given the competitiveness of the providers is purely determined by the variety and quality of the engineered exploits, they should be highly security minded, whether individual hackers or large companies who even offer security products. We expect to see best practices in protecting their code. However, even when strong protection is indeed employed, we identify some critical (and some obvious) flaws which greatly undermine the effort. In the end, we are able to seize virtually all exploit binaries offered by the root providers. In this section, based on the number of exploits, we divide the providers into three categories. From each category we will choose representative providers for detailed study, aiming to locate flaws and weaknesses in their protection methods.

As depicted in Section 4, there are three steps in the root procedure. By reversing how each step is performed, one can easily steal all exploit files and run them in any piece of malware to gain root privilege. Even the most difficult provider only took a graduate student, who is not a professional hacker, about one month of part-time work to finish, which is far less than expected for such a highly sensitive service. For reference, it took a professional Symantec research group about six months to figure out the basic structure and behavior of Stuxnet [27], which is a piece of state-sponsored malware created to attack industrial control systems. In the rest of the section, we will describe the protection methods and the weaknesses in each step for different root providers in detail.

5.1 Large Root Providers

Provider P1 (we will refer provider n in Table 2 to Pn) is one of the largest root providers currently with over a hundred exploit binaries. Its service is provided by either the PC-side program or the Android app. The most critical part in P1's architecture is an online exploit store. To update the service, P1 simply needs to add new exploits to the store. For a given device, only a selected subset is downloaded and attempted.

Protection Methods.

STEP 1: Provide device information. P1 encrypts the gathered device information such as Android device model and kernel version before sending them out to a remote server with a combination of standard encryption algorithms. Similar protection is also widely used by other large root providers to secure their online exploit stores.

STEP 2: Obtain exploits. After receiving encrypted device information, P1 servers first return a file which is an array of exploit descriptors. Each descriptor contains elaborate information about a specific exploit including an internal exploit identifier, a download link, and comments such as the affected devices. Related exploit binaries can then be fetched based on its descriptor. The descriptor file is encrypted with the same algorithm as in *STEP 1.* Besides, each file URL is encoded in a random string to prevent exhaustive crawling. A similar "descriptor file mechanism" can also be observed for P2, but with a different format.

STEP 3: Apply exploits. P1 encapsulates each exploit into a separate Linux dynamically linked shared object file (.so). These library files share a common interface of entry

point and thus can be executed, in a uniform fashion, one after another. Such files are downloaded every time when the PC-side program or the Android app is run. It is obvious that such files have to be protected in order to prevent misuse. We encounter the following: 1) The code is obfuscated by redundant instructions [33] and a custom re-arrangement procedure of the ELF binary to destroy the header and prevent disassembling. 2) A custom packing method scrambles the actual exploit code. 3) Most constant strings are encrypted. 4) There is a tamper-detection in every exploit file to ensure that the exploit can only be launched by an authentic P1 product (its own Android app), based on the app's embedded signature or the package name.

Security Flaws. Unfortunately, there exist a number of flaws that substantially undermine the strength of the protection employed by P1. We highlight them below:

- *Inconsistent protection for the same Android app obtained through different channels.* After studying P1 for a while, we realize that there are in fact two ways to get its Android app: 1) Download from P1 official website or other third-party app markets directly (Google Play prohibits such apps to be published). 2) Obtain the copy from the PC-side program's download cache. This is possible since the PC-side program will download and install the app to the connected device automatically if none is detected.

Surprisingly, the apps from these two channels behave exactly the same on mobile devices, yet there is a world of difference in their protections. The one downloaded from the official website is well protected with main "Classes.dex" encrypted and packed. This is an effective practice found in some commercial solutions (*e.g.,* Bangcle). The one obtained through the PC-side program, in contrast, does not include any protection whatsoever. Considering that Android apps tend to update frequently with only minor changes, if the core encryption logic remains the same in future versions, an attacker can misuse it for a long period of time to continually extract new exploits developed by the provider. This flaw effectively reveals all encryption algorithms used in *STEP 1* and *STEP 2*.

- *Custom obfuscation procedure leaked through online security contest held by the provider itself.* P1 employs some obfuscation methods such as a custom redundant instruction pattern and ELF header scrambling, these methods are in fact exposed in an online security contest. By simply reading the answers provided by the crowd, all details are revealed, including the obfuscation pattern and the way to restore metadata in the ELF header. Once the obfuscation is understood, remaining protections in *STEP 3* are much less effective.

- *Discrepancy in protection strength of device-side and PC-side software.* Similar to what is discussed in §4, the unprotected Android app of P1 obviate the need to deal with the PC-side program protection such as anti-debug. The opposite occurs in P3 where unprotected PC-side program enables us to ignore its well-protected Android app.

- *Leave informative names of critical functions untransformed.* Root providers often employ standard cryptographic and compression algorithms (*e.g.,* AES) to protect the code and data. However, if such obfuscation logic leaves its function and variable names untransformed (*e.g.,* a function named "md5" or a variable named "AESKey"), one can immediately recognize the algorithm and reverse the obfuscation. Such form of leakage exists in both SMALI and ARM native code of P1 and many other root providers, which undermines their protection drastically. This flaw impacts *all three steps* of P1.

- *Vulnerable tamper-detection mechanism* Signature or package name based tamper-detection can be found in many providers' exploit files. However, the detection is executed only one time at the beginning, which makes it easy to bypass — modifying one conditional jump suffices and works in all cases. Scattered and repeated tamper detection will substantially raise the protection level in *STEP 3*.

To verify that all P1's protections are successfully bypassed, we develop a piece of proof-of-concept Android malware which can fetch and run the root exploits as well as successfully obtaining root privilege on a few tested devices, including HTC One V and Sony Ericsson ST18i. In theory, this malware can leverage the full capacity of P1 since it can use all current and future exploits P1 maintains, as long as the procedure remains the same. Although we did not include the exploits that can only be launched from the PC program, they can also be downloaded and used the same way.

5.2 Medium Root Providers

We choose P4, a popular moderate-sized provider to study in this section. Different from P1, P4 stores all its exploits locally. Although there are some protection for the local store, it is overall much weaker than P1's protection. It only took us three days to obtain all P4's exploits and bypass the protection mechanism, which will be described below.

Protection Methods

STEP 1: Provide device information. Since all P4's exploits are stored locally, there is no need to send device information to any remote server. All device information is gathered locally and will then be used to guide the selection of proper exploits.

STEP 2: Obtain exploits. As soon as a specific exploit is considered proper for current device, P4 will fetch it from the local store. There are two layers of protection in this process: First, inside the Android app, MD5-based name transformation procedure is used to map an internal exploit name to the corresponding obscured file name in local store. Second, actual exploit binaries are compressed in gzip, while no informative file suffix can be seen.

STEP 3: Apply exploits. P4's exploits are all ELF executables. Similar to P1, there is also a package name based tamper-detection mechanism in each exploit binary. Besides, although no packing and obfuscation techniques are employed, all strings are encrypted and there are no informative function names.

Security Weaknesses

- *Weak protection for the device-side app.* Unlike P1, even the original apk downloaded directly from P4's official website has little protection — only some basic class and function name obfuscation is used. The major body of SMALI code is still highly readable and has given out all detailed functional and encryption/decryption logic involved in *STEP 1* and *STEP 2*. For instance, reference strings such as "md5" are not encrypted, which dramatically accelerates the reverse engineering progress.

- *Debug output turned on in ELF binaries.* The exploit binaries will output decoded strings (*e.g.,* path of a vulnerable device driver) directly to the console. Obviously the developers forgot to turn off the debug option carelessly

and this mistake significantly eases the task of locating the string decode and tamper-detection procedures. The protection in *STEP 3* is thus greatly weakened. Unfortunately, besides P4, we also find informative debug output in other providers' exploit binaries.

5.3 Small Root Providers

Small providers typically have only the device-side Android app, besides, they usually contain few but highly specialized exploits. In our survey, P6 and P7 are classified as small providers, both of their Android apps simply invokes the native exploit binary and the entire procedure has no protection. For the exploit binaries, although there are certain protections such as code obfuscation and tamper detection, they are generally primitive and easy to spot and bypass. Interestingly, the lack of effective defenses also makes it possible for larger root providers to take small providers' exploits and integrate them directly into their own products, as we observed in two larger providers.

6. CHARACTERIZATION AND CASE STUDIES OF EXPLOITS

As shown in Table 2, several top providers offer a large selection of root exploits. In this section, we dissect the 167 unique exploits from the largest provider P1 by beginning with the methodology to collect the exploits.

Exploits Collection Methodology. To download exploits from P1's online database, we need to provide sufficient information of different device models to P1's remote server (See Figure 6). Without access to a large number of real Android devices, we resort to online sources and factory images publicly available [42]. After crawling several such websites, we collect 5742 sets of device information and 2458 unique phone models with kernel ranging from 2.6.32.9 to 3.10.30 and Android version from 2.3.4 to 5.0.2. The list covers all mainstream manufacturers such as SAMSUNG, HTC, and SONY. They allow us to download 167 unique exploit binaries. Note that large providers claim to support over 10,000 or 20,000 Android device models and therefore the number of exploits we obtained may be far smaller than the actual number. Nevertheless, 167 unique exploit samples are still impressive from only 2458 phone models.

6.1 Breakdown of Exploits

Families of exploits. We hypothesize that these binaries are of high value to attackers since the number appears to be much larger than what we can find publicly. However, there is an important caveat that multiple binary files may simply be variations and adaptations of the same core exploit. In order to perform a fair comparison, we need a way to group similar binaries into families. Fortunately, the decrypted descriptor files returned from P1's server, as mentioned in Section 5.1, have an internal naming scheme to identify each exploit. An example of the internal name is *exploit98-3.2-v1*, in which "98" is used to number the exploit type, "3.2" is a specific kernel version, and "v1" indicates that this exploit is the 1st variation of an original exploit. Based on the naming scheme, we estimate that 59 different families exist, which is more than the 37 abusable public exploits still (See §2.3). From the naming schemes, we can also estimate the current size of P1's exploit families to be in the hundreds.

Based on the knowledge gained from the public exploits targeting different layers, we analyze P1's exploit binaries and its logic, *e.g.,* system calls and their parameters, we can classify a large portion of them into two main categories: 20 families belonging to kernel layer, typically featured with the use of vulnerable system calls such as `futex` (CVE-2014-3153) and `perf_event_open` (CVE-2013-2094), and 37 families belonging to driver layer, featured with operations on vulnerable device files such as /dev/exynos_mem. The remaining 2 families are difficult to classify. In the kernel layer, we have identified 17 families that can be mapped to publicly available exploits, but are unable to fully analyze the other 3. According to the exploit descriptors, we cannot locate the exploit for most affected devices from any public sources. For the driver layer exploits, we recognize 22 families as already published, but surprisingly, as we will discuss later, the remaining 15 families are potentially new exploits. Interestingly, we did not encounter any application layer exploits that typically check the existence of a process by name or a well-known file path.

New Driver-Layer Exploits. We identify 37 families of driver exploits, All driver layer exploits have the standard behavior of *open()* on a vulnerable device file in the form of */dev/file* followed by *ioctl()* or other syscalls on the file. We differentiate the device file name as kernel's built-in device or vendor-specific device and include the latter only. Even though many of them match existing exploits, we do find 15 new exploit families targeting 10 vulnerable unique device names. We are able to locate the unique device file name to specific devices which match the affected device models in P1's own exploit file descriptor. The affected devices include popular brands like SAMSUNG and some new models released less than a year ago. Due to security reasons we do not reveal the vulnerable device file names. Interestingly, in a recent research [42], it is suggested that vendor customizations of Android introduce considerable driver-layer vulnerabilities (no root exploit is discovered however). In our study, we find that such vulnerabilities can in fact lead to root privilege escalation. In retrospect, now that kernel exploits are harder and harder to come by with the latest OS security technology in place, it is natural to target the drivers to develop new exploits.

Adaptation. The most noticeable exploit family in P1's database is the one with 89 variants. By reverse engineering the exploit files we identified the family as implementations of CVE-2014-3153, the well-known "futex" kernel vulnerability. This confirms the need of adaptation of exploits as discussed in §3. To understand why this many variants are developed, we analyze the intended kernel version targeted by the 89 exploits and find 14 different kernel versions are targeted. Even for the same kernel version, P1 will apply different variants according to the kernel "build information" (*e.g.,* [#1 SMP PREEMPT Wed May 15 23:25:44 KST 2013]). The result is summarized in Table 3, P1 has covered most major linux kernel versions used by Android. For some popular versions such as "3.4.0", there exist more variants. Beside the adaptation for kernel versions, from the exploit descriptors, we also see that some variants specifically designed for certain device manufactures such as SAMSUNG and HUAWEI. The rest of the exploit families do not have many variants, *e.g.,* 42 families have only one binary.

Overall, we are impressed by the scale at which exploits are engineered by P1. It will be extremely difficult for an individual to match the amount of resources and engineering effort. We believe similar high impact kernel-level vulnera-

Kernel Version	2.6.32.9	2.6.35.7	3.0.15	3.0.16	3.0.31	3.0.8	3.4.0	3.4.39	3.4.43	3.4.5	3.4.67	3.10.0	3.10.9	3.10.30
Variants Count	2	2	1	1	3	10	21	8	1	9	2	1	1	3

[1] The calculation is based on our own collection, actual amount of P1's variants may be larger.

Table 3: P1's adaptation for CVE-2014-3153 (based on kernel version)

bility such as CVE-2015-3636 (currently pioneered as PingPong Root) may be of similar value and require substantial adaptation effort. It is reported that the root exploit affects many latest Android devices such as Samsung Galaxy S6 and HTC One (M9) and the list is growing.

Timeline to add new exploits. As we have shown, exploits from large commercial root providers largely overlap with publicly available ones. One important metric indicating the competitiveness of the providers is the time it took from the date original exploits were first published to the date that they are incorporated. Even though there is no comprehensive data, we do have a unique data point on the latest PingPong root exploit, which was first published in May 2015. The same exploit is incorporated in P1 roughly two to three months later. We note that PingPong root is techinically intrisnic and involved. It is impressive that the provider has finished reverse engineering, developing, and testing of the exploit within such a short period. In fact, the incorporation happened before the full technical detail of the exploit is released and any proof-of-concept code is available. This demonstrates that commercial root providers are capable and swift, which is another reason why they may become targets of attackers.

6.2 Interaction with Advanced Security Mechanism

In 109 out of 167 P1's exploits (including all "futex" exploits), we observe special treatment for SELinux, which forms the base of the advanced Android security mechanisms such as SEAndroid [37] and Knox [17]. To support fine-grained mandatory access control, SELinux introduced the concept of "security context" whereby a process running as root may still be subject to restrictions imposed by the policies on the "context" it is running as. This effectively eliminates the powerful root in the traditional Linux. However, in AOSP, SELinux policy in Android 4.4 and below generally make the app domain either permissive or unconfined. Unless customized by vendors such as Samsung, it means that SELinux is effectively "disabled". Furthermore, even when SELinux policies are configured to be enforcing, as is done since Android 5.0 (AOSP), kernel-layer exploits can subvert SELinux easily by overwriting the related kernel data structures, given SELinux operates under the assumption that the kernel is intact. Specifically, almost in all cases of the 109 exploits, they overwrite not only the uid but also the sid and osid so that the security context effectively becomes "init", which is the most privileged one and is able to access almost all system resources. After that, they will write to */proc/self/attr/current* to change its string representation of security context to "u:r:init:s0".

Similarly, we also observe modifications to Linux "process capability" related kernel data structures such as $cap\_effective$ to 0xFFFFFFFF. Since process capabilities in Linux share the same threat model as SELinux, it is not hard to imagine that they can be subverted in the same way.

When compared with public proof-of-concept exploit source code, we rarely find such level of care in dealing with the additional restrictions set by SELinux and process capabilities.

	AVG	Lookout	Norton	Trend Micro
Original	N	N	N	N
Unpacked	N	N	N	13
Bypassed	N	N	N	N

[1] N:No threat detected

[2] All anti-virus at newest version when testing.

Table 4: Anti-Virus Test Results on P1's exploits

6.3 Anti-Virus Test

Since the root exploits are highly sensitive and may be leveraged by various Android malware, it is expected that anti-virus software on Android platform can identify most of them, including the ones implemented by root providers. We select the same 4 representative Android anti-virus products to test P1's 167 exploits. Because originally downloaded exploits from P1's database have packed the actual exploit code and employed a tamper-detection mechanism, we crafted 3 different versions for every exploit: 1) Original exploit fetched directly from P1's servers, with packing and tamper-detection on. 2) Unpacked exploit, which will expose all actual exploit logic to anti-virus products. 3) Re-packed exploit with tamper-detection disabled. The last version can be highly dangerous since it can be utilized freely by malware that can unpack and execute at run time.

To test the antivirus software, we embed all exploit files of one version at a time in a self-developed Android app to trigger the proper scan. We test one antivirus software at a time. If a message is prompted that the app is safe to open, it indicates the antivirus software fails to detect any exploit in the app, and we uninstall the antivirus and install the next. If an alert is given flagging our app as malicious, we attempt to identify which subset of exploits are flagged by embedding one exploit file at a time and retest.

The test results are shown in Table 4. It is disappointing to see that no packed exploit is detected by any antivirus software. It is likely due to the custom obfuscation implemented by the provider that is not recognized. However, even for the unpacked ones, only Trend Micro can recognize 13 out of 167 exploit files as malicious. It is worth mentioning that the highly dangerous futex exploits as well as the PingPong root exploit are not caught by any antivirus software. This contrasts with the result in §2.4 where the public PingPong root exploit is in fact detected by Trend Micro. This suggests 1) P1's implementation of PingPong root is sufficiently different; 2) Trend Micro uses some kind of signature-based detection that is not effective at catching variants of the same exploit. Overall, the result shows that the state-of-the-art security products on Android platform still cannot address root exploits effectively. Worse, packing and obfuscation can easily evade detection.

7. RELATED WORK

Android malware analysis. Android malware detection and analysis attracted much attention of research in the past few years [43, 32, 44, 22]. The Android Malware Genome Project [43] has offered a public dataset of Android malware from year 2010 to 2012. The analysis covers behaviors from privacy invasion, financial charges, remote control,

to root exploits embedded in the malware. In the ANDRUBIS [32] project, a more recent malware set of 400K samples collected between 2012 and 2014 is examined. It gives a more up-to-date view, yet little discussion is on malware carrying root exploits. Other sources include Contagio minidump [3] and VirusTotal [20]. DroidRanger [44] and DREBIN [22] attempt to detect Android malware by leveraging a carefully selected set of features.

So far, no evidence has shown that a single piece of malware embeds a large number of root exploits, likely because of the engineering challenge, *e.g.,* many exploits need to be adapted to work on specific device models. In our study, it is alarming to see the potential that malware can abuse the root provider logic to achieve this goal.

Android root exploits and defense. While a comprehensive characterization of Android root exploits is lacking, point studies have shown that root exploits were indeed abused by malware in the wild [44, 43]. As described in the Android Malware Genome Project [43], 36.7% of 1260 malware samples had embedded at least one root exploit.

Root providers present a unique position in computer history that they legitimately collect and distribute a large number of fresh root exploits. In theory, all commercial root providers should provide adequate protections on the exploits. In practice, unfortunately, as long as one of the providers fails to achieve that, malware authors can successfully "steal" the well engineered, adapted, and tested exploits against a diverse set of Android devices.

The development of Android comes with much improvement in Security. SeAndroid was shown to be effective against many root exploits that target user-level applications [37]. Research proposals use dynamic behaviors such as system calls and other events to detect root exploits [44, 29, 35]. Unfortunately, none of them is bullet-proof. For instance, new exploits such as TowelRoot and PingPong Root are not impacted by SeAndroid since they exploit kernel-layer vulnerabilities directly. In addition, the more expensive dynamic analysis techniques require root privilege to operate which limits their applicability and not to mention its impact on battery life.

Reverse engineering and anti-reverse engineering. Anti-reverse engineering aims at transforming a program into a semantically equivalent one yet much more difficult to comprehend and reverse engineer [25, 24]. Encryption, packing, symbol stripping, instruction reordering, *etc.* are commonly used obfuscation techniques against reverse engineering [24]. The key used to encrypt the code can either be embedded directly in the binary or burned onto the hardware [23, 38]. Most programs simply embed the key directly in its binary, including the famous STUXNET [27] and the binaries in root providers, as there is little support from the hardware to encrypt and decrypt instructions on the fly in general-purpose computing systems [38]. In addition, a different program binary encrypted with a different key needs to be distributed for every machine, which can be costly and complex to manage. More advanced techniques such as VM-based software protection [23] also exist. They dramatically increase the cost of reverse engineering by employing custom instruction set architectures.

In response to such anti-reverse engineering and obfuscation techniques, deobfuscation techniques are also developed [31, 36, 23]. It is not clear when such arms race will end as fundamentally the code under protection has to run physically on a machine controlled by the adversary. In a recent work, Zeng *et al.* proposes to use trace-oriented programming to implement binary code reuse [41]. The idea is that as long as the execution trace is observed and recorded at runtime, they can be extracted and reused. In principle, such ideas can be applied to extract obfuscated root exploit code. However, it is not sufficient as the self-verification logic still needs to be identified and removed.

On Android, the situation is not much different except some advanced obfuscation tools such as custom-VM-based protections are not yet available. As most large root providers need to protect both PC-side software and device-side app, the obfuscation strength is determined by the weaker side.

8. DISCUSSION AND CONCLUSION

Ethics. The study on root providers can be controversial. We study them because of two reasons: 1) root provider is a unique product in history that has unique characteristics. 2) Although legitimate, the functionality is implemented by exploiting vulnerabilities of the target system, which presents significant security risks. The goal of our research is to understand and characterize the risk that well-engineered exploits from the root providers can be stolen and easily repackaged in malware. By studying the protection mechanisms employed by root providers, we aim to quantify their strength and point out areas of weaknesses.

To protect the root providers, in the paper, we intentionally anonymize their names when detailed results are shown. Further, we plan to release our findings to the corresponding root providers and device vendors.

Android-side root vs. PC-side root. In this study, we cover in detail mostly the root exploits implemented directly on Android devices. Most large root providers in fact offer both PC-side as well as Android-side root methods. The reason we focus on Android-side root is its risk of being abused by malware. It is worth mentioning that, as demonstrated by recent studies [39], a compromised PC can infect the mobile devices connected to it. Under such threats, the PC-side root exploits also become dangerous and are subject to abuse by PC malware. We leave this for future study.

Conclusion. In this paper, for the first time, we uncover the mysterious Android root providers. We find they not only make significant efforts to incorporate and adapt existing exploits to cover more devices, but also craft new ones to stay competitive. However, these well-engineered exploits are not well protected, it is extremely dangerous if they fall in the wrong hands. This may also trigger a public policy/legal discussion on whether to regulate such companies that manufacture up-to-date exploits that are freely distributed.

Acknowledgments

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

9. REFERENCES

[1] Android Vulnerabilities – All vulnerabilities. http://androidvulnerabilities.org/all.html.

[2] Beating up on Android. http://titanium.immunityinc.com/infiltrate/archives/Android_Attacks.pdf.

[3] Contagio minidump. http://contagiominidump.blogspot.com.

[4] CVE-2014-3153 aka towelroot. https://github.com/timwr/CVE-2014-3153.

[5] Don't Root Robots: Breaks in Google's Android Platform. https://jon.oberheide.org/files/bsides11-dontrootrobots.pdf.

[6] Exploit DB database. https://exploit-db.com/.

[7] How To Root An AT&T HTC One X. http://rootzwiki.com/topic/26320-how-to-root-an-att-htc-one-x-this-exploit-supports-185/.

[8] iRoot, Retrieved on May 10, 2015. http://www.mgyun.com/m/en.

[9] It's Bugs All the Way Down. http://vulnfactory.org/.

[10] One Click Root for Android, Retrieved on May 10, 2015. http://www.oneclickroot.com/.

[11] Rage Against the Cage. http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz.

[12] Razr Blade Root. http://vulnfactory.org/public/razr_blade.zip.

[13] Root Genius, Retrieved on May 10, 2015. http://www.shuame.com/en/root/.

[14] Root the Droid 3. http://vulnfactory.org/blog/2011/08/25/rooting-the-droid-3/.

[15] [Root] ZTE z990g Merit (An avail variant). http://forum.xda-developers.com/showthread.php?t=1714299.

[16] [Root/Write Protection Bypass] MotoX (no unlock needed). http://forum.xda-developers.com/moto-x/orig-development/root-write-protection-bypass-motox-t2444957.

[17] Samsung Knox. https://www.samsungknox.com/.

[18] TacoRoot. https://github.com/CunningLogic/TacoRoot.

[19] Virus Profile: Exploit/MempoDroid.B. http://home.mcafee.com/virusinfo/virusprofile.aspx?key=1003986.

[20] VirusTotal. https://www.virustotal.com/.

[21] Xoom FE: Stupid Bugs, and More Plagiarism. http://vulnfactory.org/blog/2012/02/18/xoom-fe-stupid-bugs-and-more-plagiarism/.

[22] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.

[23] A. Averbuch, M. Kiperberg, and N. Zaidenberg. Truly-Protect: An Efficient VM-Based Software Protection. *Systems Journal, IEEE*, 2013.

[24] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical report, The University of Auckland, 1997.

[25] C. S. Collberg and C. Thomborson. Watermarking, Tamper-proffing, and Obfuscation: Tools for Software Protection. *IEEE Trans. Softw. Eng.*, 2002.

[26] J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski. *Android Hacker's Handbook*. Wiley, 2014.

[27] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. Technical report, Symanetic, 2011.

[28] D. Guido and M. Arpaia. *The Mobile Exploit Intelligence Project*. Blackhat EU, 2012.

[29] Y. J. Ham, W.-B. Choi, and H.-W. Lee. Mobile Root Exploit Detection based on System Events Extracted from Android Platform. In *SAM*, 2013.

[30] X. Hei, X. Du, and S. Lin. Two Vulnerabilities in Android OS Kernel. In *ICC*, 2013.

[31] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proc. of USENIX Security Symposium*, 2004.

[32] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *BADGERS*, 2014.

[33] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM CCS*, 2003.

[34] OpenSignal. Android Fragmentation Visualized. http://opensignal.com/reports/2015/08/android-fragmentation/, 2015.

[35] Y. Park, C. Lee, C. Lee, J. Lim, S. Han, M. Park, and S.-J. Cho. RGBDroid: A Novel Response-Based Approach to Android Privilege Escalation Attacks. In *LEET*, 2012.

[36] R. Rolles. Unpacking Virtualization Obfuscators. In *WOOT*, 2009.

[37] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS*, 2013.

[38] J. I. Torrey. HARES: Hardened Anti-Reverse Engineering System. Technical report, Assured Information Security, Inc., 2015.

[39] T. Wang, Y. Jang, Y. Chen, S. Chung, B. Lau, and W. Lee. On the Feasibility of Large-Scale Infections of iOS Devices. In *Proc. of USENIX Security Symposium*, 2014.

[40] W. Xu. *Ah! Universal Android Rooting is Back*. Blackhat, 2015.

[41] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation Resilient Binary Code Reuse Through Trace-oriented Programming. In *ACM CCS*, 2013.

[42] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *IEEE Security and Privacy*, 2014.

[43] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Security and Privacy*, 2012.

[44] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS*, 2012.